Arithmetic instructions

The MIPS assembly language notation

```
add a, b, c
```

(integer)

instructs the processor to add two variables b and c, and to put their sum in a.

The following sequence of instructions adds four variables b, c, d, and e

```
add a, b, c     // a ← b + c
add a, a, d     // a ← b + c + d
add a, a, e     // a ← b + c + d + e
```

Another example:

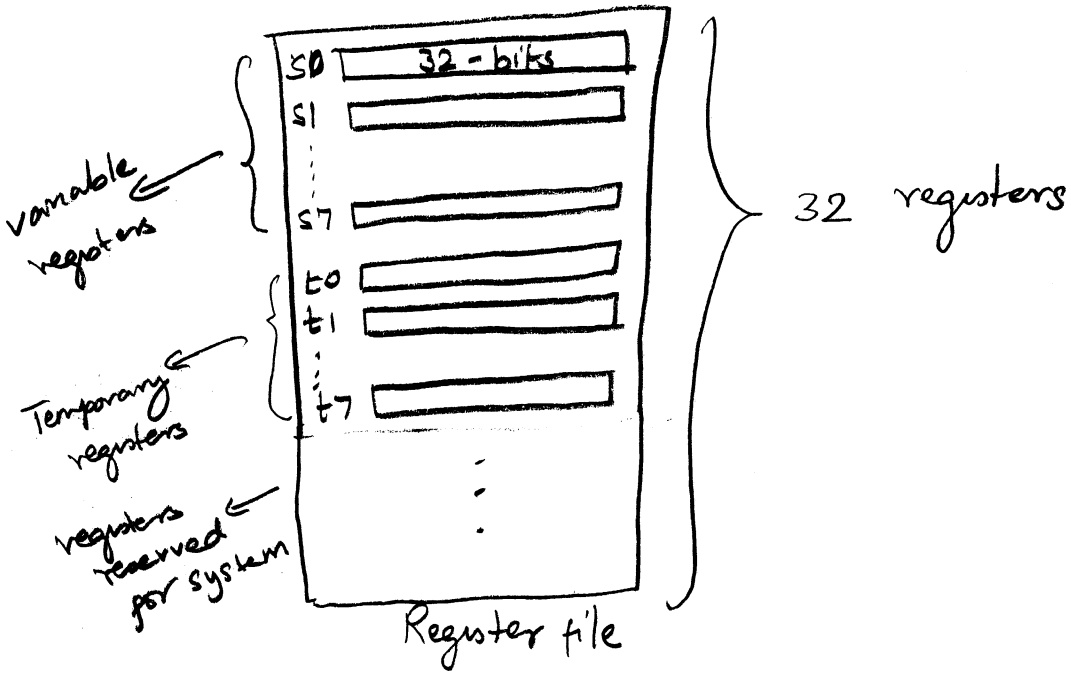
$$f = (g+h) - (i+j)$$

The compiler produces:

```
add t0, g, h    // Temporary variable t0 contains g+h
add t1, i, j    // Temporary variable t1 contains i+j
sub f, t0, t1   // f ← t0 - t1
```

Assignment of variables to registers

A computer has a "register file" comprising multiple registers. Let us assume that each register can store a word (32 bits).



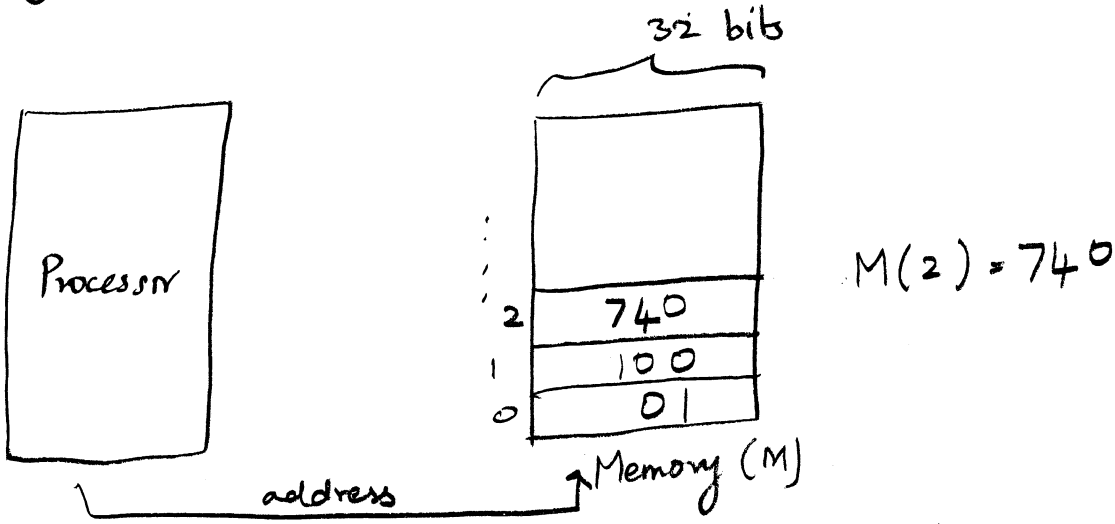
$f = (g + h) - (i + j)$, gets compiled to:

Assume f is assigned to $s0, g to $s1, h to $s2, i to $s3, and j to $s4.

```
add  $t0, $s1, $s2   // $t0 ← g+h
add  $t1, $s3, $s4   // $t1 ← i+j
sub  $s0, $t0, $t1   // f ← $t0 - $t1
```

Data transfer

Consider memory to be a large, single-dimensional array, with the address acting as the index to that array, starting at 0.



How does a compiler translate this assignment statement?

A is an array of 100 words

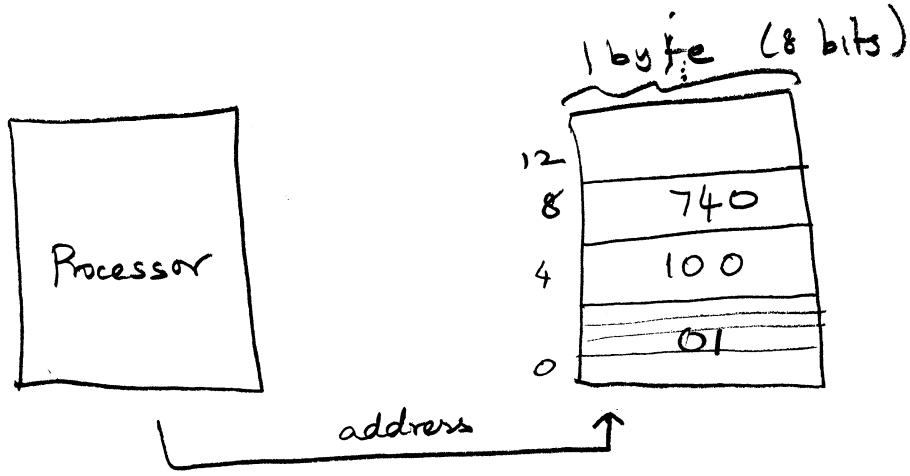
```
g = h + A[8] ;
```

Let g be in $S1; h be in $S2.
Let us assume that the starting address (or base address) of A is in $S3.

offset, base register

```
lw   $t0, 8($S3)   // t0 ← A[8]
add  $S1, $S2, $t0 // g ← h + A[8]
```

What if memory was organized as bytes?




```
A[12] = h + A[8];
```

Assume h is associated with $s2.
Base address of A is in $s3.

```
lw  $t0, 32($s3)   // A word is 4 bytes.
add $t0, $s2, $t0  // $t0 ← h + A[8]
sw  $t0, 48($s3)   // stores h + A[8] into
                      A[12].
```

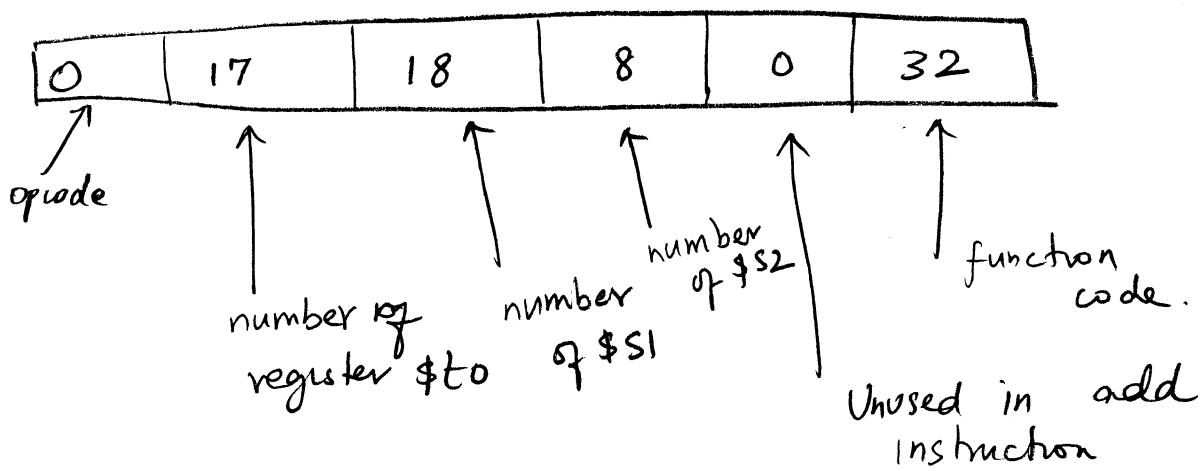
Representing Instructions in the computer

Instructions in assembly code must be converted to binary.

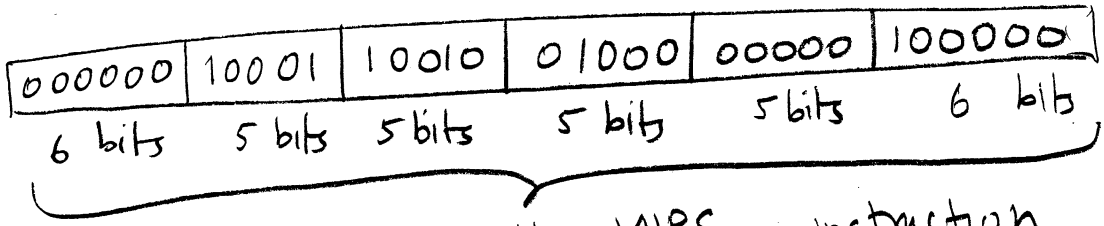
Example :-

```
add $t0, $s1, $s2 // MIPS assembly code
```

Using the MIPS ISA handout, we can translate the above instruction as

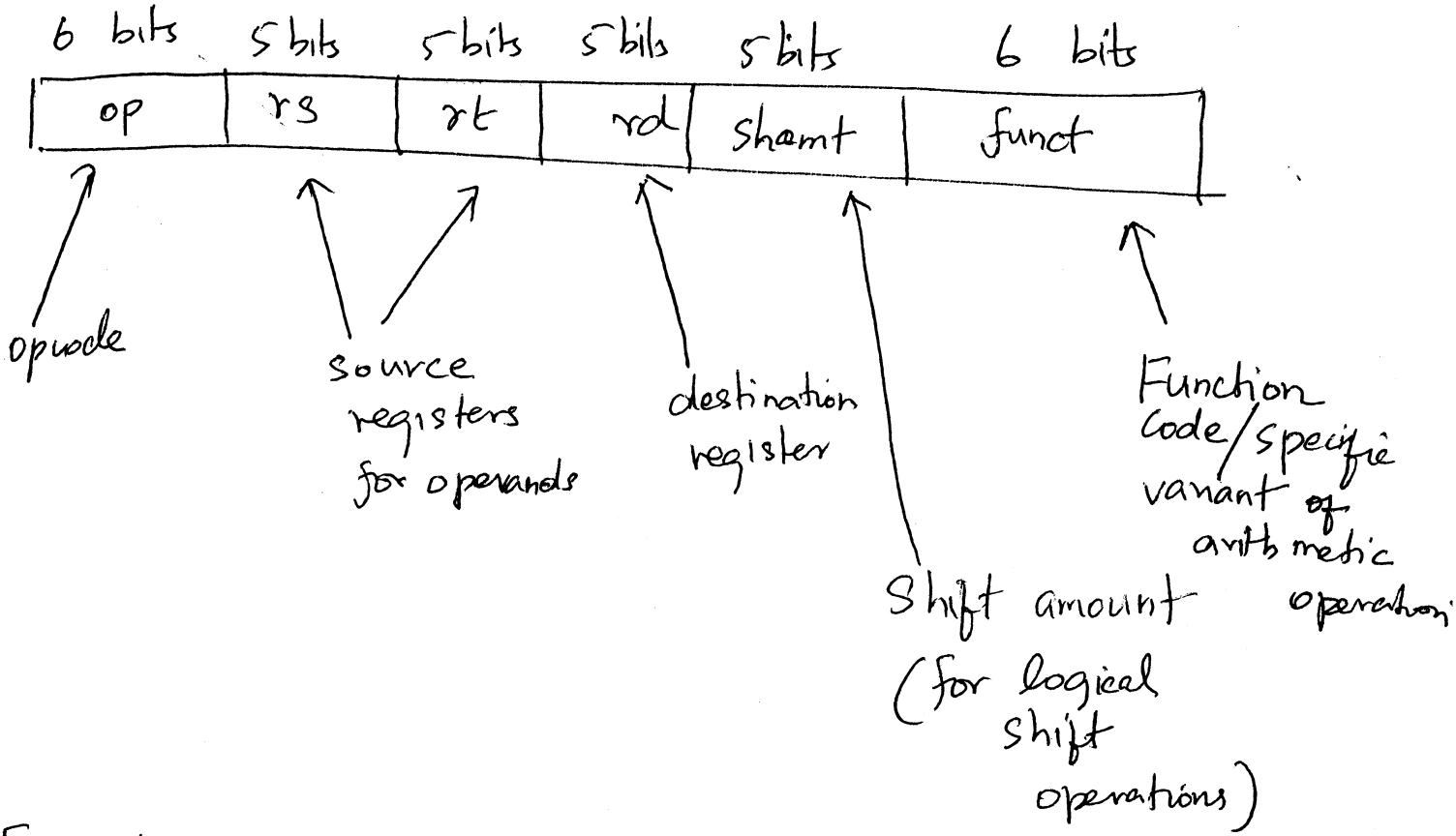


The corresponding binary code :



32-bit MIPS instruction, in machine language.

Format of an arithmetic (or R-type) instruction.



Example:

```
add $s1, $s2, $s3   // $s1 = $s2 + $s3
```

destination register

source registers

Translating MIPS assembly code to Machine language.

Example:

Consider the following C code:

```
A[300] = h + A[300];
```

$t1 has the base address of array A and $s2 has the variable h.

The assembly code is:

```
lw   $t0, 1200($t1)   // Memory is byte addressable
add  $t0, $s2, $t0    //$t0 gets h + A[300]
sw   $t0, 1200($t1)   // stores h + A[300] back into A[300]
```

MIPS machine language code (consult your ISA handout)
(in decimals)

op	rs	rt	rd	address/shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Logical Operations

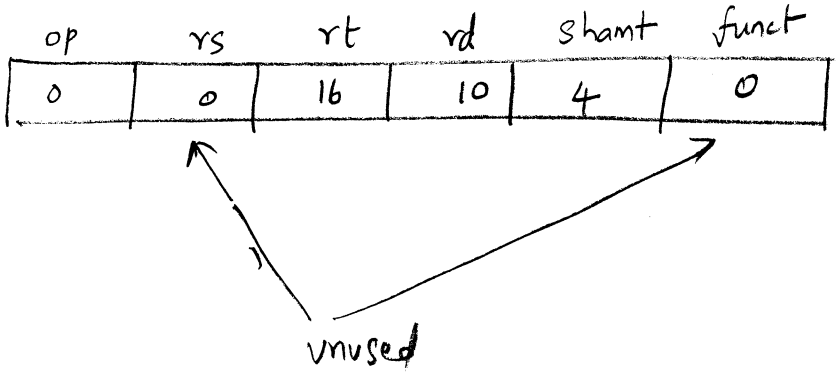
- Shift left (<<) denoted by sll in MIPS assembly
- Shift right (>>) denoted by srl in MIPS
- Bit-wise AND (&) denoted by and, andi
- Bit-wise OR (|), denoted by or, ori
- Bit-wise NOT (~), " " nor

immedia

The shamt field in the R-format sepecifies the shift amount. So,

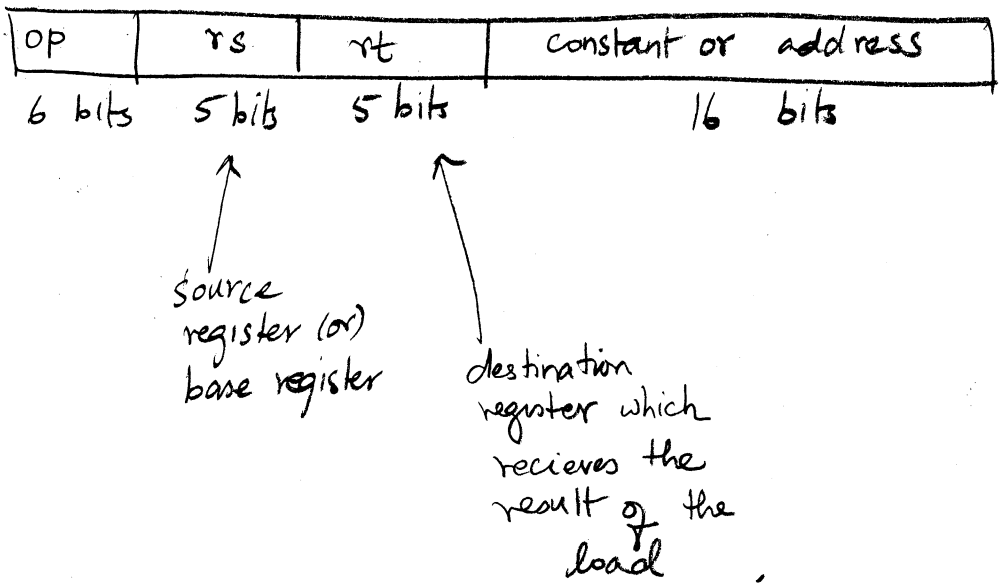
```
sll  $t2, $s0, 4   // register $t2 = $s0 << 4 bits.
```

is translated as




```
and  $t0, $t1, $t2   // $t0 = $t1 & $t2
or   $t0, $t1, $t2   // $t0 = $t1 | $t2
```

Format for data transfer and immediate instructions or I-format.



Instructions for Making Decisions

The MIPS assembly language includes two decision-making instructions:

beq register1, register2, L1

bne register1, register2, L1

Means go to the statement labeled L1 if the value in register1 equals the value in register2. "beq" stands for branch if equal.

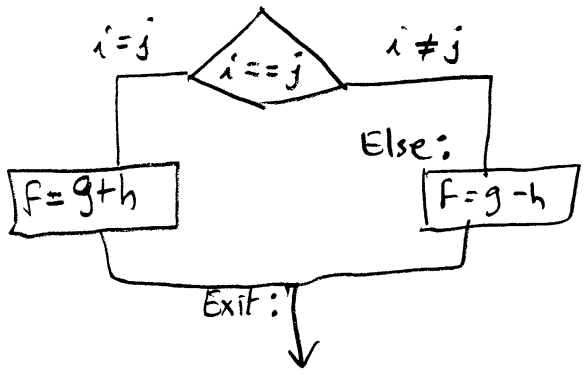
Means go to the statement labeled L1 if the value in register1 does not equal the value in register2. "bne" stands for branch if not equal.

Example :-

c-code

```
if (i==j) f=g+h;
else
    f=g-h;
```

Let the variables f through j correspond to five registers $s0 to $s4. First, consider this flow chart



The MIPS code is:

```
        bne  $S3, $S4, Else  // Goto Else if i≠j
        add  $S0, $S1, $S2   // f=g+h (Skipped if i≠j)
        j Exit               // Unconditional jump (j) statement
Else:   sub $S0, $S1, $S2
Exit:
```

The assembler relieves the programmer from the tedium of calculating addresses for branches.

Loops:

```
while (save[i] == k)
    i = i + 1;
```

Let i and k be in registers $S3 and $S5. The base address of the array save is in $S6.

```
Loop: sll $t1, $S3, 2  // Remember that memory is byte addressable. So to index into i, we need to multiply by 4, or left-shift by 2
      add $t1, $t1, $S6 // $t1 = address of save[i]
      lw  $t0, 0($t1)   // $t0 = save[i]
      bne $t0, $S5, Exit // The loop test. Goto Exit if save[i] ≠ k
      addi $S3, $S3, 1  // i = i + 1
      j Loop
Exit:
```

Sometimes, it is useful to test if a variable is less than another variable. For example,

```
if (i > j)  i = j
```

The MIPS instruction provides a "set on less than" or slt. For example,

```
slt  $t0, $s3, $s4
```

means that register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4; otherwise $t0 is set to 0.

The instruction,

```
slti $t0, $s2, 10
```

means that $t0 is set to 1 if $s2 < 10; or $t0 is set to 1.

slti is an immediate version of the slt instruction.

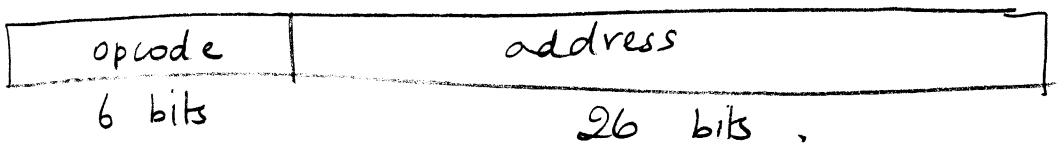
Also, the register $zero is a special register and always has the value 0.

Finally, unconditional branches are denoted by the "j" instruction.

```
j  L
```

means that jump to target address L.

The unconditional jump is a "j-type" instruction with the following format:



Supporting procedures in computer hardware

Consider the following procedure/function call:

```
int foo (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

To execute the above procedure, the compiler must:

1) Place the parameters g, h, i, and j in a place where the procedure can access them.
2) Transfer control to the procedure
3) Allocate space for the local variable f
4) Perform the task
5) Place the result in a place where the calling program can access it.
6) Return control back to the calling program.

MIPS has the following registers to support procedures.

$a0 – $a3 : Four argument registers to pass parameters.

$v0 – $v1 : Two value registers in which to return values.

$ra : One return address register to return to the point of origin.

MIPS also has an instruction just for procedures. "jump and link" instruction "jal".

jal Procedure_address

jumps to the address specified in Procedure_address and simultaneously saves the value of PC + 4 in $ra. ← remember that memory is byte addressable!

The "link" is stored in $ra and is called the return address.

MIPS also supports a "jump register" instruction.

```
jr $ra
```

means an unconditional jump to the address specified in a register.

What to do if the compiler needs more registers for a procedure than the 4 argument and 2 return value registers?

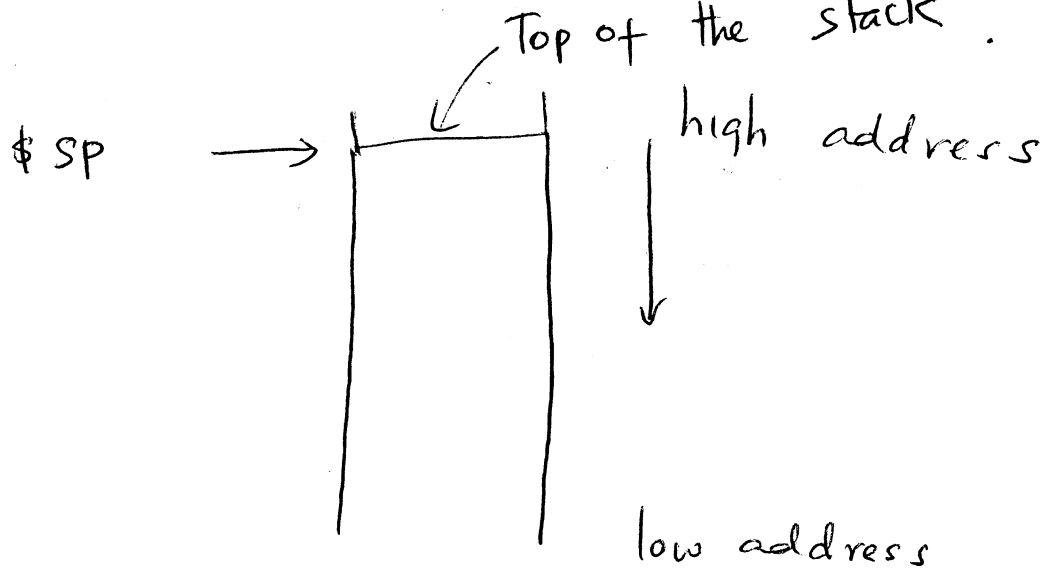
Key issue: Any registers needed/used by the caller program must be restored to the values contained before the procedure was called.

Solution — Use a stack to save register values that must be restored after the procedure is ~~called~~ finished.

Stack → a last-in-first-out queue.

MIPS hardware provides a register to store the stack pointer ($sp), or the starting address of the stack.

Stack operations



A stack "grows" fro higher addr to lower addresses.

push: Placing data into the stack (decrement $sp)
pop: Removing data from the stack (increment $sp)

Let us return to our example:

The parameter variables g, h, i, and j correspond to argument registers $a0, $a1, $a2, and $a3. f corresponds to $s0.
The label of the procedure is foo:

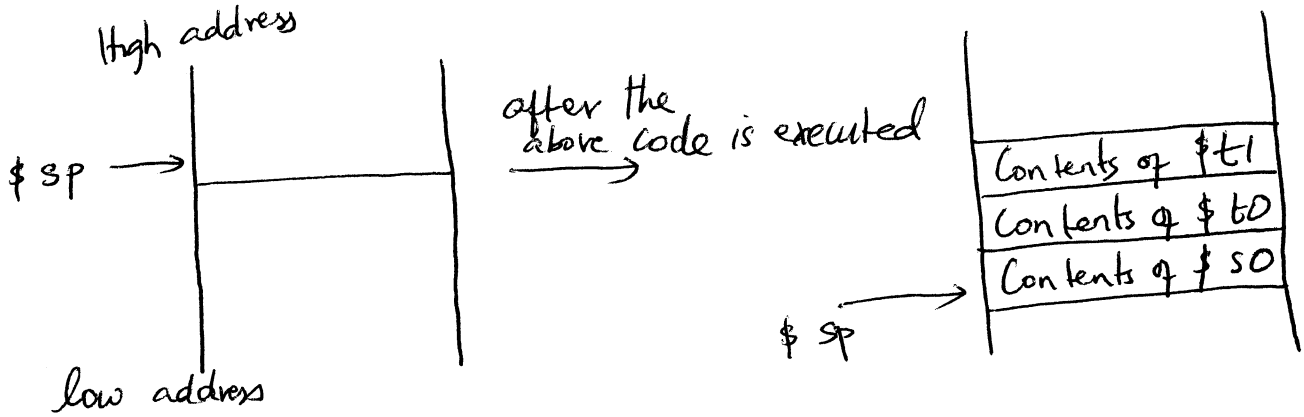
Step 1:- Save the registers used by the procedure.

If (g+h) is placed in $t0,
(i+j) is placed in $t1, and
f is placed in $s0,

then we must save the previous values of $t0, $t1, and $s0.

We push the old values of $s0, $t0, and $t1 into the stack.

```
addi  $sp, $sp, -12  // Adjust the stack to
                        make room for three
                        items
sw    $t1, 8($sp)    // Save $t1
sw    $t0, 4($sp)    // Save $t0
sw    $s0, 0($sp)    // Save $s0
```




```
add   $t0, $a0, $a1  // $t0 = g+h
add   $t1, $a2, $a3  // $t1 = i+j
sub   $s0, $t0, $t1  // f = $t0 - $t1.

add   $v0, $s0, $zero // places f in the
                         return value register
                         $v0.
```

Before returning, we restore the three old values of the registers saved by popping them from the stack.

```
lw $s0, 0($sp)      // restore $s0 for the caller program
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12   // Adjust the stack to delete 3 items

jr $ra              // Jump back to calling routine
```